***Build 2 Architecture Design Document***

**Introduction**

Our project’s second build mainly focuses on few of the core functionalities such as attack phase and card exchange. Observer pattern and dice roll functionality are also implemented.

The design pattern that we used is Model-View-Controller. The reason for using MVC design pattern is that it allows different programmers to work on different business logic simultaneously. This makes the work faster and thereby saves time. Our build is filled with multiple functionalities, and keeping in mind its compatibility to Extreme Programming, we decided to use Incremental approach. By using this approach, small working releases were produced that were tested at unit level before integrating with one other.

**Development Model**-Agile (Incremental)

**Design Pattern**- MVC

**MVC Architecture**

It is an architecture pattern that separates the application into 3 components- the model, the view, the controller. Each component is built to handle specific aspects of an application.

**Model**: It corresponds to all data related logic that the user works with. It represents the data that is being transferred between view and controller components. Controllers act as an interface between Model and View components to process all the business logic and incoming requests, manipulate data using the Model component and interact with the Views to render the final output.

In our project Build 2, Model contains class files.

* Player- It contains player methods and fortification, attack and roll dice methods. This class uses observer pattern, when data is changed it notifies the observers.
* Map- It contains continent list and territories list to store map files.
* Territory-It holds the territory details such as name of the territory, continent and player it belongs to.
* Continent-It contains the territory list belonging to that continent.
* GameDetails- It contains map validation method and is used for creating map (reading map file and storing continents and territories), creating player objects, player colors, distributing armies to players and territories and for creating cards.
* Card- It contains card type and territory name.

**View**: It is used for all user interface logic of application. This depends on data received from model or sometimes from controller.

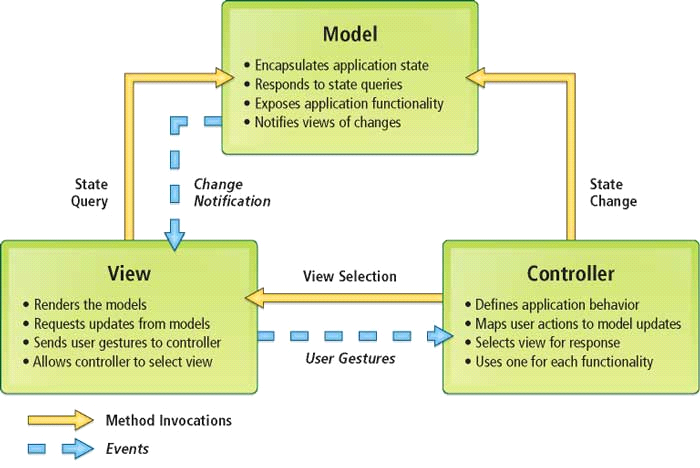
In our project Build 2, View contains class files.

* Sample-It will display main user interface where the user has to choose map editor, new map or new game.
* Newmap- It will display new map window containing text area, save button and cancel button. This user interface is for creating new map and giving name of new map and saving it.
* MapEditor- This is the user interface for loading the map and editing and saving the map file.
* Loadgame- This interface loads the map and displays the armies on the map and player stats and it has buttons for managing phases.
* CardExchange- Interface to display the type of cards the player holds and to exchange them.
* Phase- Interface to know the actions happening in the game.
* PlayerDomination- Interface to display the player control percentage over the map, number of armies and continents the player holds.

**Controller**: Controllers act as an interface between Model and View components to process all the business logic and incoming requests, manipulate data using the Model component and interact with the Views to render the final output.

In our Project Build 2, Controller contains class files.

* Main- Controller to handle the main user interface (sample.fxml).
* MapEditor- validates the map after editing and saves the map.
* NewMap- Validates the map and saves with the user provided names.
* LoadGame- Handles the game interface, like giving control to the nest player, displaying player stats and changing mouse clicks based on game phases.
* PlayerDomination- contains methods to add the text to the player domination interface using observer pattern.
* PhaseView- contains methods to display the game actions using observer pattern.
* CardExchange- Displays all the card types to the cards exchange interface. Implements card exchange logic when user wants to exchange cards.



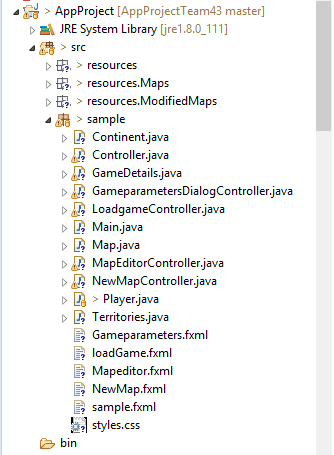
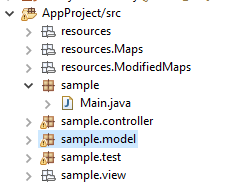
* **MVC Architecture**

**Refactoring-**

Refactoring is a controlled technique for improving the design of an existing code base. Its essence is applying a series of small behavior-preserving transformations, whose cumulative effect is quite significant. By doing them in small steps we reduce the risk of introducing errors.

Some examples of how we applied refactoring in Build 2 are :

1. Organizing files into separate packages i.e. keeping all view files in View package, all controller files in Controller package and all model files in Model package to mimic MVC architecture, which was not done in Build1.

**Build 1 Build 2**

1. Modifying already existing function logic to support various game phases. In build 1 we used mouse click only to increment armies’ count of territory, but in build2 we changed the logic of mouse click function such that now it will increment armies when the game is in startup phase, and returns the territory name when you click on attack phase and in fortification phase.



In the above code we can see the use of switch cases for different phases of the game. So mouse click functionality changes according to the game phases.

1. Removing some unnecessary lines of code.

**Extreme Programming features used:**

* Planning-

A general plan for all the 3 builds was made and a detailed plan for the current build was built on the basis of which the whole team started the work.

* Sustainable pace-

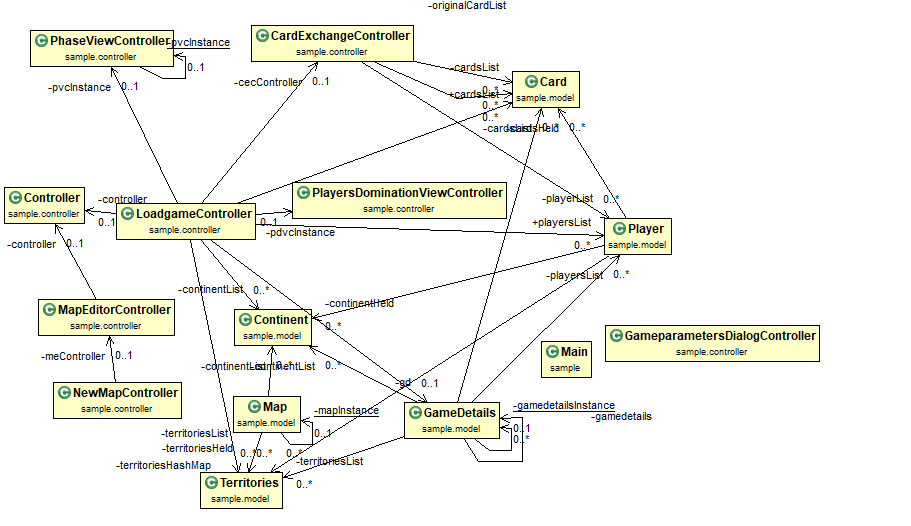
We all have been meeting regularly and discussing the work that is to be done. We have been helping each other in difficult and time-consuming parts of the project to reduce the pressure on a particular team member and ensure timely completion of a particular part of the build.

* Small Releases-

We developed and deployed partially working application with certain features for every release regularly.

* Collective Ownership-Everyone in the team was allowed to make changes in any part of the code. Any team member could fix the errors in the code by just informing about the changes to the member who made the code.

**UML Diagram**

****

* **Class Diagram**

**References**

* <https://www.whizsolutions.co.uk/advantages-using-mvc-framework-web-development/>
* <https://www.tutorialspoint.com/extreme_programming/extreme_programming_additional_features.htm>
* <https://moodle.concordia.ca/moodle/pluginfile.php/3506605/mod_resource/content/1/Lecture%202.pdf>
* <https://www.wikihow.com/Play-Risk>
* <https://howdoyouplayit.com/risk-how-do-you-play-risk/>
* <https://www.hasbro.com/common/instruct/risk.pdf>